CHAPTER 6
PC HOST SOFTWARE

In this chapter we'll write two example PC host programs, both of which will set us up for writing our application program in Chapter 7.

- **Example 1:** This will be a **USB Device Display** program that extracts and displays all of the descriptors, introduced in the previous chapters, from all of the devices currently attached via the USB bus. The program does this by searching for devices on the USB bus, so we'll learn valuable techniques for the "low-level" USB activity. The example is implemented in two versions: 1B is written in Visual Basic and 1C in Visual C++ (there is no 1A!)
- **Example 2:** This will be an **HID Display** program that searches for and displays information on the currently installed human interface devices (HID). The program uses a high-level file-style access to the I/O devices so that we focus on the **information** transferred between the PC host and the I/O device and not on the details within the USB packets. Example 2 is also implemented in two versions: 2B and 2C.

The source code and project files of both are on the CDROM so that you can compare the solutions and expand them as you prefer.

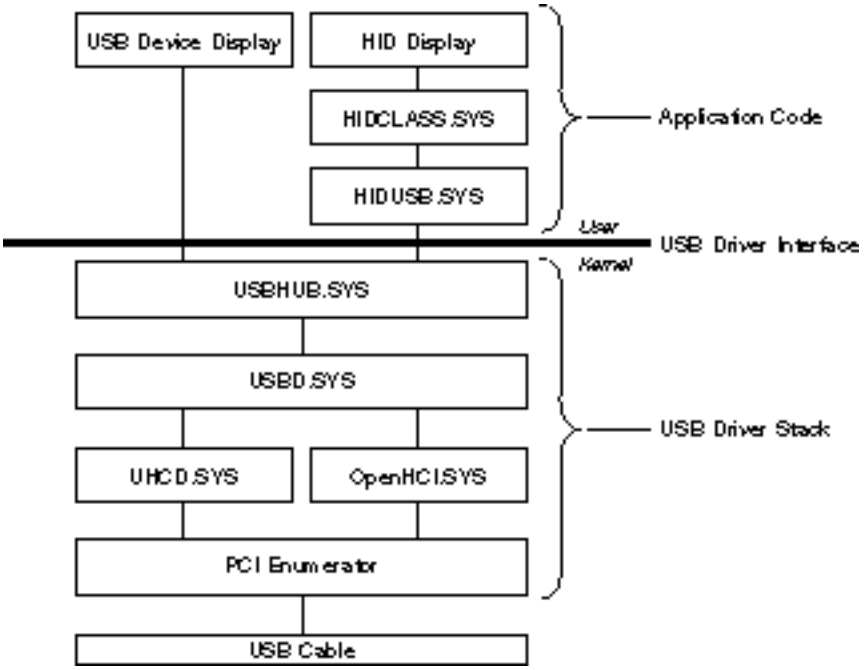

Figure 6-1. USB support in Windows is layered

The USB Driver Interface of the Windows operating system uses the Win32 Driver Model (WDM) layered architecture. Our device display program will interface directly to the kernel-level USB driver stack, and our HID display program will interface with the HID Class driver stack. Both example programs make extensive use of system calls; these are declared in a separate module to allow the programs to be more easily understood. The filter drivers will be installed above USBD.SYS and will have visibility into all of the operating system calls.

The example programs will use many of the Windows system libraries and declaration files. You should download the Windows 2000 DDK from www.microsoft.com /hwdev to gain access to these files. The DDK is free.

VISUAL BASIC REVIEW

Visual Basic allows intuitive human interfaces to be rapidly prototyped. There are a few things about Visual Basic that I should repeat here for your convenience.

A Visual Basic program is built around a graphical human interface that uses objects such as buttons, graphics, and text (Figure 6-2). The visible window that you interact with during program construction is called a FORM, and a single program could include many forms to present information to the user. The user interacts with the program by clicking the mouse when it's over an object on the form or by entering text from the keyboard.

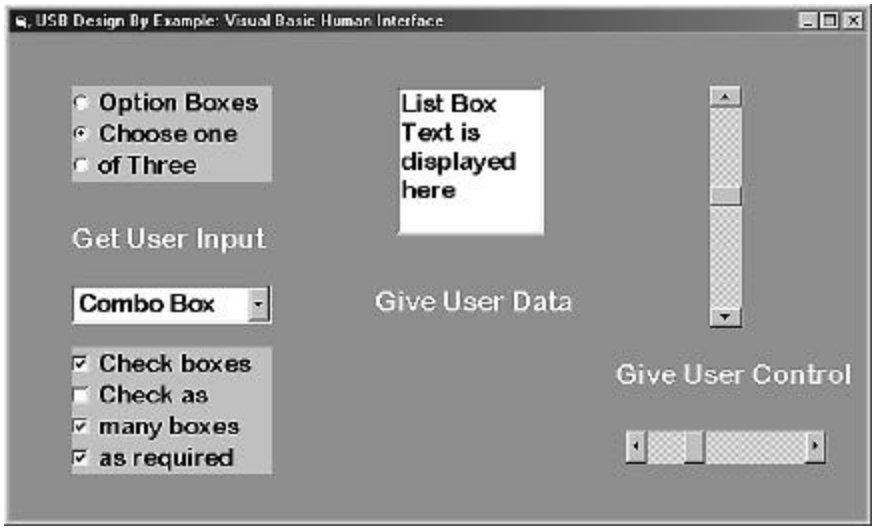

Figure 6-2. Visual Basic has a graphical human interface

Each object has many properties that can be set or tested. When one of these properties changes, Visual Basic first checks to see if you have defined an "action" subprogram, or script, to run in response to this interaction. A BUTTON object, for example, will be informed when the mouse is over it and if one of the mouse buttons is clicked. Visual Basic does all of the hard work for you, letting you focus on a few responses to actions on key visual objects.

Many application programs make extensive use of the ListBox feature of Visual Basic. A ListBox can be displayed as a multiline list (with scroll bars if required) or as a single line with the list contents available in a drop-down menu. Figure 6-3 shows both styles.

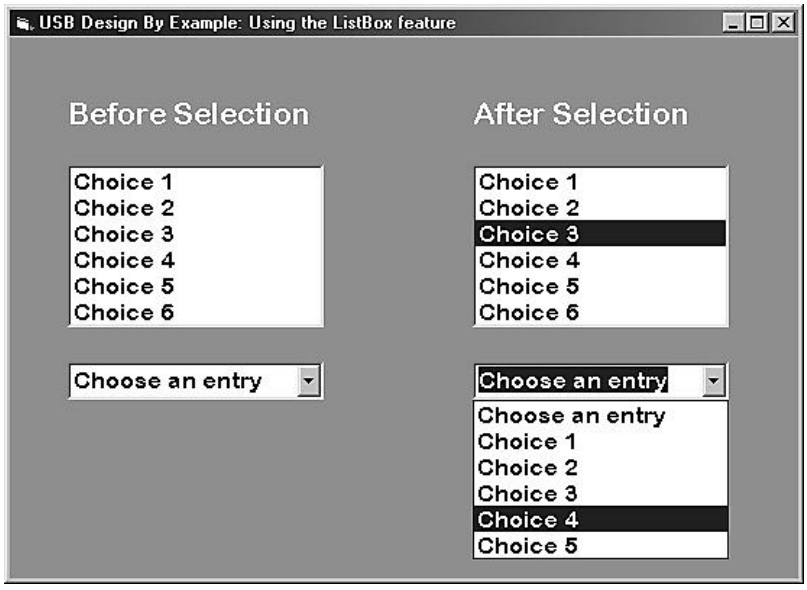

Figure 6-3. Two styles of ListBox

The two styles of ListBox allow related visual information to be quickly assimilated by the user. The programmer has a lot of control over the displayed elements, and Visual Basic generates an "event" if the user clicks on any of the entries of a multiline list or chooses an entry from a drop-down menu.

A ListBox has two data values associated with each line: The displayed text is called a **LIST (n)** value, and there is a hidden element called **ITEMDATA (n)**. This feature allows programmer-useful information to be coupled with user-useful information that, in turn, simplifies the example software.

Other key Visual Basic attributes are described briefly below.

Variable Names—The size of all variables is declared by adding a symbol after a variable's name:

- "&" means "32-bit integer"
- "%" means "16-bit integer"
- "$" means "a string"

There are others, but I have not used them in these example programs.

User-Defined Data Types—Complex data types can be constructed using the base data types supported by Visual Basic. We saw in the previous chapters that large data structures are used to describe the functions and features within USB. These data structures can be defined as user-defined data types, and this allows our software to be written more simply and clearly.

Parameter Passing—All of the OS libraries that we will be calling are written in C and expect parameters to be passed by VALUE. The convention that Visual Basic uses is to pass parameters by REFERENCE, i.e., the **address** of the variable. C programs often include pointers to variables, and the programmer can manipulate pointers as easily as manipulating data variables. Visual Basic does not allow this—using pointer arithmetic, it is as easy to write buggy code as it is to write efficient, elegant code! Careful declaration of the called libraries will implement any parameter conversions necessary. Where this parameter conversion could not be finessed via the function or subroutine declaration, I have used an "AddressFor" function supplied with Dan Appleman's *Visual Basic 5.0 Programmers Guide to the Win32 API* (ISBN 1-56276-446-2). This library is supplied, with permission, in the Chapter 6 directory on the CD-ROM.

System Call Parameter Values—Microsoft provides a lot of scope and flexibility with their operating system libraries. With this flexibility comes responsibility: If you pass an invalid parameter to one of these libraries, the results are not always predictable. I have limited the amount of damage that can be done by using "magic" numbers, typically hexadecimal values, in these calls. **DO NOT CHANGE THESE** unless, of course, you know what you are doing. (I used standard Microsoft equates in the Visual C++ versions of the programs – this gives you a little more insight into the system calls).

Many books have been written on API programming, and my goal is not to explain the benefits and dangers of these functions but to USE THEM to implement USB-focused applications.

There are many textbooks available for Visual Basic, and I recommend getting a good understanding before proceeding further in this chapter.

EXAMPLE 1B: USB DEVICE DISPLAY

Because USB supports the dynamic attach and detach of I/O devices, the address at which a particular device is located will vary according to the order in which it is attached and which other devices are already attached. Thus, we can't rely on a static address to locate an I/O device; instead, we must work with the operating system and gain access to its system tables. The Windows operating system provides system calls to allow us to do this.

We must first locate the root of the Plug and Play I/O subsystem and then search for USB host controllers. USB host controllers have a predeclared name of "HCDx," where "x" is a digit and identifies a unique host controller. The OS can support multiple USB host controllers, and each can support up to 126 devices on its shared bus. Systems that use several high-bandwidth I/O devices may have multiple host controllers, and these must be interrogated too.

A PC system may also have a mix of high speed and full speed host controllers – this is transparent at this software layer.

Once a USB host controller is found, its corresponding root hub is identified. The ports of this hub are then queried to discover if a device or another hub is connected. If another hub is discovered, then the ports on that are queried. All hubs are queried recursively to the maximum depth of five as defined in the USB Specification. Figure 6-4 shows the algorithm used to create a list of attached USB devices.

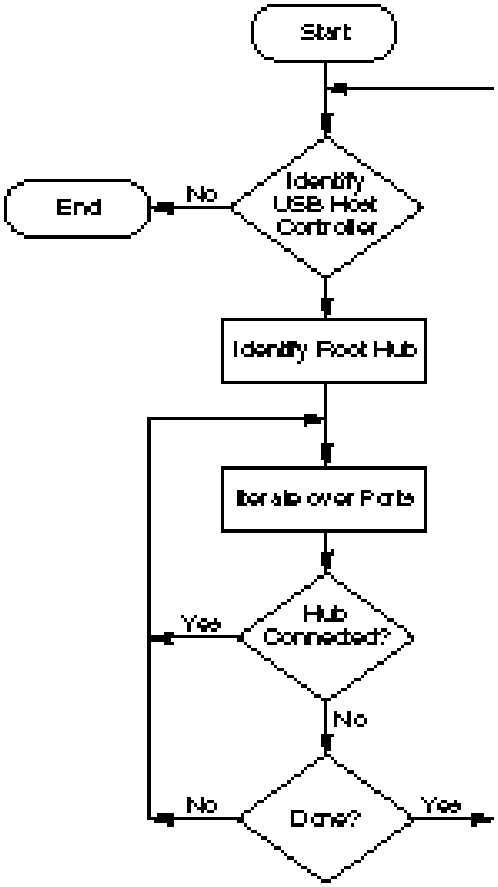

Figure 6-4. Search algorithm for USB devices

Example 1B—Step 1: Human Interface Design

The first part of any Visual Basic program is the design of the user's view, or Human Interface, of the program. I chose to use two forms—the first to display the topology of the attached USB devices and the second to display the descriptors of a chosen device. Figure 6--5 shows the opening form for this example. It consists of four buttons labeled Host Controller 0 through Host Controller 3, a Status Line, and a Display area.

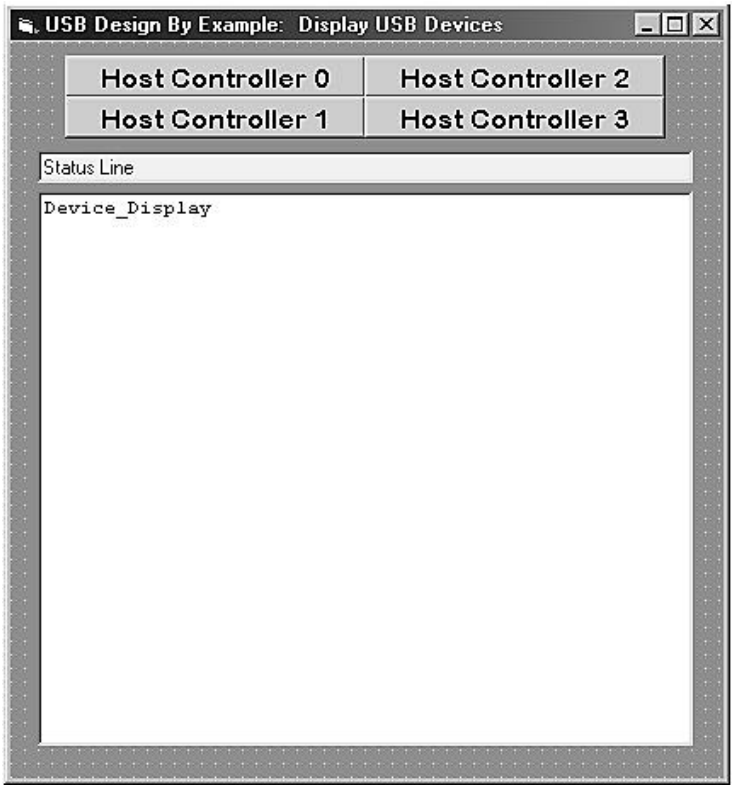

Figure 6-5. The first, and opening, form

Figure 6-6 shows the second form; it is designed to look as similar to the descriptor diagrams in the previous chapters as possible. All devices will have a **Device** descriptor and at least one **Configuration** and one **Interface** descriptor. A device may have one or more **Endpoint** descriptors and may have one or more **Class** descriptors (such as HUB, Human Interface Device, Communications Device). This human interface design allows one of each descriptor type to be displayed at a time; a CHOICE box above each descriptor type selects one if multiple descriptors are detected on this device. The CHOICE box also allows the names of each descriptor parameter to be displayed. As an aid to learning, if any parameter name of any descriptor is clicked, the program displays the corresponding parameter value. The final CHOICE box displays the **String** descriptors if present on this device.

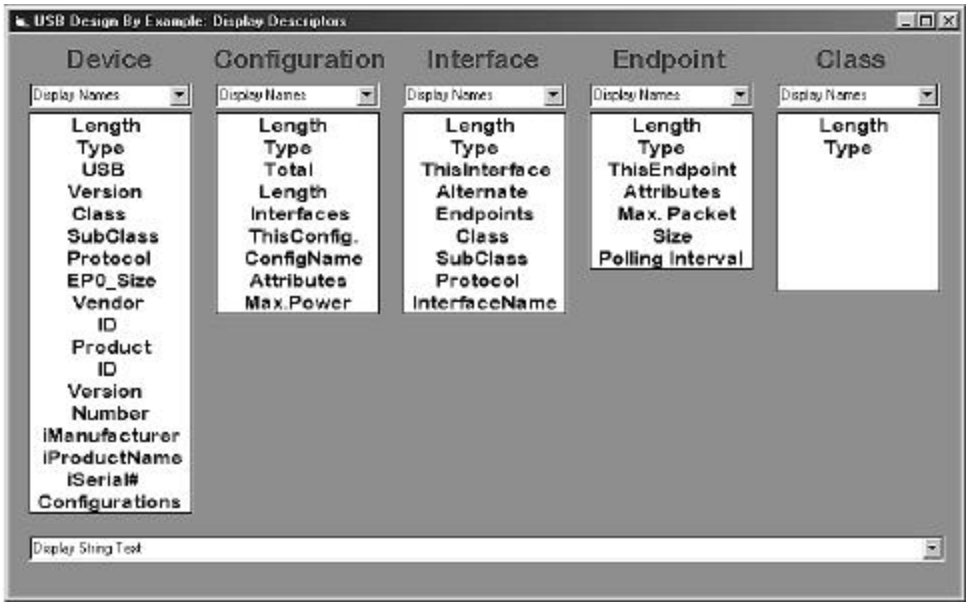

Figure 6-6. Descriptor display form

Example 1B—Step 2: Initializing Program

As the program loads, Visual Basic sends a "loading form" event message to the program. If a subroutine called FORM_LOAD is present, then this will be executed. Our example uses this subroutine to search for USB host controllers as shown in Figure 6-7. The example searches for up to four controllers—if you expect more, then add more "Host Controller" buttons. The buttons are declared as an array [HCD (3)], so no new software will be required for these additional buttons.

If a host controller is found, the program changes the background color of the corresponding button to green and enables that button. A system identifier for the host controller is stored in the button's TAG field for later use. If a host controller is not found, the program changes the background color of the corresponding button to orange and disables the button. The result of this FORM_LOAD subroutine is an indication of how many host controllers are present and a prompt to the user to select one to study.

```
'  Look for Host Controllers.
'   I limit the search to 3. There may be more but this is unlikely.
'    The Host Controller Buttons are HCD(0) to HCD(3)
'   Try opening the controller using its Symbolic Name
'
For ControllerIndex& = 0 To 3
    HostControllerName$ = "\\.\HCD" & ControllerIndex&
    HostControllerHandle& = CreateFile(HostControllerName$, &H40000000, 2, 0, 3, 0, 0)
    If HostControllerHandle& > 0 Then
        HCD(ControllerIndex&).Tag = HostControllerHandle&
        HCD(ControllerIndex&).BackColor = RGB(0, 256, 0) 'Green = GO
        HCD(ControllerIndex&).Enabled = True
        Else
        HCD(ControllerIndex&).BackColor = RGB(256, 128, 0) 'Amber = wait
        HCD(ControllerIndex&).Enabled = False
        End If
    Next ControllerIndex&
StatusBox.Text = "Select a Host Controller"
```

Figure 6-7. FORM_LOAD identifies host controllers

Example 1B—Step 3: Choosing a Host Controller

This step does all of the USB-literate work in this program, so we'll move more slowly here. Clicking on a host controller button will start the USB device data collection process. Many operating system calls will be made, and I chose to encapsulate each of these in a separate subroutine. Having separate routines enables error-checking following each call, and the routine returns only if there are no errors. The approach allows for precise local error-checking, and the calling routine does not need to check for errors.

A Windows I/O subsystem consists of many nodes. These could be USB devices, SCSI devices, hard disk controllers, PCI bus connectors, or many others. The operating system refers to all I/O devices by their *node **detail** information* and their *node **connection** information*—this may appear verbose for our USB example, but remember that the I/O subsystem has to handle all I/O devices in a consistent way. Each I/O device family is connected to a **root node**, so our first task is to identify the USB root node.

Windows maintains symbolic names for all of the I/O devices it supports. We saw in "Example 1B—Step 2: Initializing Program" that the root node name for a USB host controller was "HCDn." Using a standard system call, we can ask the operating system for the name of the node at the next level down in the I/O device hierarchy. We can repeat this process and thus identify all of the connections to all of the nodes. We are, in fact, traversing the tree structure of the I/O devices starting at the root.

Example 1B—Step 3.1: Identifying the Root Hub Node

We have an operating system handle for the host controller (obtained in Form_Load and stored in the button's TAG field)—now we can request the name of the root hub using a DeviceIoControl system call. The operating system requires a two-stage process to get this name (Figure 6-8). The first system call returns the LENGTH of the name and the second returns the name in UNICODE format. The GetNameOf$ subroutine also converts the Unicode format into Visual Basic string format. The system name is a very long, you can print it our if you like. The example program uses this name in the next step of this process.

```
Function GetNameOf$(DeviceName$, DeviceHandle&, API_ID&)
Dim NameBuffer As UNameType
'
'   First need to get the length of the name string
Status& = DeviceIoControl(DeviceHandle&, API_ID&, 0, 0, NameBuffer.Length, 260, BytesReturned&, 0)
If Status& = 0 Then ErrorExit ("Could not get LENGTH of " & DeviceName$ & " Name")
If NameBuffer.Length > 256 Then ErrorExit (Name$ & " Name > 256 Characters")
'
'   . . . and then the string. It will be returned in UNICODE format
Status& = DeviceIoControl(DeviceHandle&, API_ID&, NameBuffer.Length, NameBuffer.Length, _
NameBuffer.Length, NameBuffer.Length, BytesReturned&, 0)
If Status& = 0 Then ErrorExit ("Could not get TEXT of " & DeviceName$ & " Name")
temp$ = "": i = 0   'A simple unicode to basic string conversion
Do While NameBuffer.UnicodeName(i) <> 0
temp$ = temp$ & Chr(NameBuffer.UnicodeName(i)): i = i + 2: Loop
GetNameOf$ = temp$
End Function


'   Get the name of the host controller
HostController$ = GetNameOf("Host Controller", HCD(Index).Tag, &H220424)
StatusBox.Text = "Host Controller: " & HostController$ & " selected"
'
'   Get the name of the Root Hub and open a connection to it
RootHubName$ = GetNameOf("Root Hub", HCD(Index).Tag, &H220408)
```

Figure 6-8. Getting the root hub name

We can now make a connection to the root hub using a CreateFile system call. This operating system call returns a handle we can use to discover information about this node. We'll use a DeviceIoControl system call to retrieve this information, and Figure 6-9 shows the node information returned. Important for our traversal of the device tree is RootHubNode.NodeDescriptor.PortCount, because this defines the number of connection points for other nodes.

Because we may want to get more information on this connection later, we store key parameters in a large data table. Note that we could not use the Listbox.ItemData feature because there is more than a single variable.

```
Public Type HubDescriptor
   Length As Byte:     HubType As Byte:    PortCount As Byte:     Characteristics(1) As Byte
   PowerOn2Good As Byte: MaxCurrent As Byte: PowerMask(63) As Byte: End Type

Public Type NodeInformation
   NodeType As Long: NodeDescriptor As HubDescriptor: HubIsBusPowered As Byte: End Type
```

Figure 6-9. Node information data structure

Example 1B—Step 3.2: Probing Root Hub Connections

Now that we have identified a set of ports to the root hub, we request NodeConnectionInformation from the operating system so we can identify what, if anything, is connected to each port of the hub. A DeviceIoControl system call is used to retrieve this information, and Figure 6-10 shows the connection information returned. Information relevant for our traversal of the device tree is USBDeviceInfo.ConnectionStatus and USBDeviceInfo.DeviceIsHub. We use this information to decide if we need to reiterate on the ports of another hub.

```
Public Type DeviceDescriptor
   Length As Byte:    DescriptorType As Byte:    USBSpec(1) As Byte:    Class As Byte
   SubClass As Byte:  Protocol As Byte:          MaxEP0Size As Byte:    VendorID(1) As Byte
   ProductID(1) As Byte:       DeviceRevision(1) As Byte:       ManufacturerStringIndex As Byte
   ProductStringIndex As Byte: SerialNumberStringIndex As Byte: ConfigurationCount As Byte: End Type

Public Type NodeConnectionInformation
   ConnectionIndex As Long: ThisDevice As DeviceDescriptor: CurrentConfiguration As Byte
   LowSpeed As Byte: DeviceIsHub As Byte: DeviceAddress(1) As Byte: OpenEndPoints(3) As Byte
   ThisConnectionStatus(3) As Byte: MyEndPoints(29) As EndPointDescriptor: End Type
```

Figure 6-10. Connection information data structure

Again, key information is logged into our USBDeviceInformation data table, and the device display is updated. Figure 6-11 shows the subroutine that implements the node traversal and data collection. This GetPortData function calls itself if it discovers that a hub is connected to one of the ports. No special programming is required in Visual Basic for this recursive operation, because all subroutines and functions automatically support recursion.

The GetPortData function will eventually complete when all branches of the tree have been traversed.

```
Function GetPortData&(Handle&, PortCount As Byte, HubDepth&)
Dim ThisDevice As Byte

For PortIndex& = 1 To PortCount
   Call GetNodeConnectionData(Handle&, PortIndex&)

   ThisDevice = 0 ' default value, no device connected
   PortStatus& = DeviceData(DataIndex).ConnectionData.ThisConnectionStatus(0) ' save some typing!
   If PortStatus& = 1 Then
      ThisDevice = DeviceData(DataIndex).ConnectionData.DeviceAddress(0)
      DeviceData(DataIndex).DeviceHandle = Handle&
      End If
   ' Create an indented display so that Hubs and their connections are easily seen
   Indent$ = " ": For i& = 1 To HubDepth&: Indent$ = Indent$ & ".": Next i&
   DeviceName$ = ThreeDecimalCharacters$(ThisDevice) & Indent$ & "       Port["
   Mid$(DeviceName$, 10) = ":"

   If PortStatus& <> 1 Then ' There is not a valid device on this port, tell user
      Device_Display.AddItem DeviceName$ & PortIndex & "] = " & ConnectionStatus$(PortStatus&)
      Else ' have a Device or a Hub connected to this port

      If DeviceData(DataIndex).ConnectionData.DeviceIsHub Then
'
'  Need to discover how many ports are supported on this hub.
'  Follow the same proceedure as we did for the root hub = get it's name, "open" it and get the node info.
         ExternalHubName$ = GetExternalHubName(PortIndex&, Handle&)
         ExternalHubHandle& = OpenConnection(ExternalHubName$)
         Call GetNodeInformation(ExternalHubHandle&)
         DeviceData(DataIndex).DeviceType = 2 'Hub
'  LAST thing we do is update the display status of this device connection
         Device_Display.AddItem DeviceName$ & PortIndex & "] = Hub Connected"
'
'  Discover what, if anything, is connected to the ports of this Root Hub
         Level& = GetPortData&(ExternalHubHandle&, _
DeviceData(DataIndex - 1).NodeData.NodeDescriptor.PortCount, HubDepth& + 1)

      Else 'we have a device connected to this port
         DeviceData(DataIndex).DeviceType = 3 'IODevice
         Device_Display.AddItem DeviceName$ & PortIndex & "] = IO Device Connected"
         End If 'USBDeviceInfo.DeviceIsHub
      End If 'PortStatus& <> 1
   Next PortIndex&
End Function
```

Figure 6-11. GetPortData function collects device data

Example 1B—Step 4: Descriptor Display

The result of Step 3 is a completed USBDeviceInformation data table and a completed Device_Display. If the device list is longer than the display space allocated, then Visual Basic automatically creates scroll bars so that all of the device entries can be viewed.

The user selects a device by clicking on an entry in the Device_Display ListBox. The Device_Display_Click subroutine identifies the selected entry and passes control to the Display_Descriptor module.

All of the data we have accessed up to now has been provided from operating system tables. We now need to collect the descriptor information for the selected device, and this will involve sending USB requests to the selected device.

Example 1B—Step 4.1: Collect Device Descriptor Information

Request Packets (Figure 6-12) must be created and sent to the USB device. The same process is used to send any type of request: A packet is preformatted and a DeviceIoControl system call is made. To retrieve a Configuration Descriptor, for example, two calls must be made: The first retrieves just the 18-byte configuration descriptor, and the second uses the TotalLength parameter to read all of the descriptors. If the selected device has multiple configurations, then the data for each must be retrieved. Figure 6-11 shows the subroutine that implements this data collection.

```
Private Sub CollectDescriptors(Selected&)
' Collect all of the descriptors from the selected device and store them in the DescriptorData byte array
' Start with the Device Descriptor
For i& = 1 To 18: DescriptorData(i&) = DeviceData(Selected&).ConnectionData.ThisDevice.Contents(i& - 1): Next i&
Nexti& = 18
' Now get local copies of some key variables
Dim Configuration As Byte: Dim StringIndex As Byte
Handle& = DeviceData(Selected&).DeviceHandle
ConnectionIndex& = DeviceData(Selected&).ConnectionData.ConnectionIndex
ConfigurationCount = DeviceData(Selected).ConnectionData.ThisDevice.Contents(17)
For Configuration = 1 To ConfigurationCount
   TotalLength& = GetConfigurationDescriptor(Handle&, ConnectionIndex&, Configuration - 1)
' Copy the Configuration Descriptor into the DescriptorData byte array
For i& = 1 To TotalLength&: DescriptorData(Nexti& + i&) = PCHostRequest.ConfigurationDescriptor(i& - 1): Next i&
   Nexti& = Nexti& + TotalLength&: Next Configuration
' Check for Strings
StringIndex = 0
Do While TotalLength& <> 0
   TotalLength = GetStringDescriptor(Handle&, ConnectionIndex&, StringIndex)
   StringIndex = StringIndex + 1
   For i& = 1 To TotalLength&: DescriptorData(Nexti& + i&) = PCHostRequest.ConfigurationDescriptor(i& - 1): Next i&
   Nexti& = Nexti& + TotalLength&: Loop
End Sub
```

Figure 6-12. Retrieving all of the descriptor data

Example 1B—Step 4.2: Interpret the Configuration Descriptor

Step 4.1 returns a large, unformatted buffer of bytes that is a concatenation of the device descriptors. Rather than just display this raw data, the ParseDescriptor subroutine (Figure 6-13) interprets the data and extracts indexes into the buffer; these indexes are stored in the CHOICE ListBox.ItemData entries for easier display. The format of the descriptor data, with an initial LENGTH byte and a subsequent TYPE byte, makes the data parsing quite straightforward.

```
Private Sub ParseDescriptors()
DeviceCount = 0: ConfigurationCount = 0: InterfaceCount = 0: EndpointCount = 0: ClassCount = 0
Index& = 1
Do While DescriptorData(Index&) <> 0
   Select Case DescriptorData(Index& + 1) ' What TYPE of descriptor is this?
   Case 1
   DeviceCount = DeviceCount + 1: Choice(0).AddItem "Display Values"
   Choice(0).ItemData(Choice(0).ListCount - 1) = Index&
   Case 2
   ConfigurationCount = ConfigurationCount + 1: Choice(1).AddItem "Configuration " & ConfigurationCount
   Choice(1).ItemData(Choice(1).ListCount - 1) = Index&
   Case 3
   If StringCount <> 0 Then Choice(5).AddItem "String " & TwoHexCharacters$(CByte(StringCount)) & _
                                                  " = " & GetString$(Index&)
   StringCount = StringCount + 1
   Case 4
   InterfaceCount = InterfaceCount + 1
   Choice(2).AddItem "Interface " & ConfigurationCount & ":" & InterfaceCount
   Choice(2).ItemData(Choice(2).ListCount - 1) = Index&
   Case 5
   EndpointCount = EndpointCount + 1
   Choice(3).AddItem "Endpoint " & ConfigurationCount & ":" & EndpointCount
   Choice(3).ItemData(Choice(3).ListCount - 1) = Index&
   Case Else ' Must be a Class Descriptor
   ClassCount = ClassCount + 1
   Choice(4).AddItem "Class(" & TwoHexCharacters$(CByte(DescriptorData(Index& + 1))) & ") " _
                                                  & ConfigurationCount & ":" & ClassCount
   Choice(4).ItemData(Choice(4).ListCount - 1) = Index&
   End Select
   Index& = Index& + DescriptorData(Index&)
   Loop
' Fill out the default data for the Descriptors
For i% = 0 To 4
   If Choice(i%).ListCount = 1 Then ' this descriptor type is not present so remove it from the display
      Choice(i%).Visible = False: Descriptor(i%).Visible = False
      Else ' fill with data
      Choice(i%).ItemData(0) = Choice(i%).ItemData(1): Call AddDescriptorData(i%, 0)
      End If
   Next i%
If StringCount = 0 Then Choice(5).Visible = False
End Sub
```

Figure 6-13. Parsing the device descriptor information

Example 1B—Step 4.3: Displaying Individual Descriptors

The descriptor display initializes with all text entries in the display boxes. The individual descriptors are listed in the CHOICE box above each display box as previewed in Figure 6-6. Individual text entries can be replaced by their value by clicking on an entry, or the whole descriptor data can be displayed by choosing its entry from a CHOICE box. Figure 6-14 shows a typical display following several user interactions.

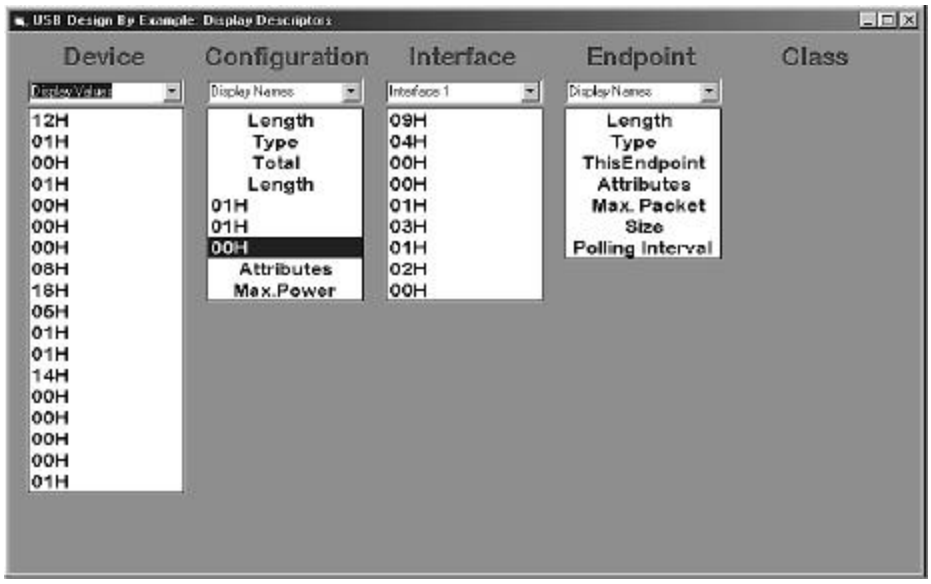

Figure 6-14. Descriptor display in operation

Example 1B—Summary

The example demonstrated the code required to identify all of the USB Devices attached to a PC host. A specific device was identified. The example listed all devices, and a user selected one. Other programs could search through the collected information and choose a device based on some identifying criteria. Note that the device descriptor, which contains VendorID, ProductID, and Version#, is contained within the NodeConnectionInformation data structure, so searching on these parameters is straightforward. A targeted request packet was sent to the identified device. The example requested all of the device descriptors, but your program could send any valid request—the program structure is the same. "Low-level" passing of request packets was successfully demonstrated.

EXAMPLE 1C: USB DEVICE DISPLAY

This example is the same function as example 1B except that it is implemented in Visual C++. Many readers of the first edition requested this! I shall follow the same steps as 1B and where possible I shall use the same function names, variable names and comments so that it will be easy to compare the two examples.

I will be using the Visual C++ application wizard to develop a Win32app.

<< Cover the C++ version of the first example here >>

<<This segment will be added later>>

EXAMPLE 2B: HID DEVICES

The second example is simpler because we need to extract and display only a single system table. Windows will keep all HIDs in a single table regardless of their source (multiple USB controllers, PCI bus, or legacy ports). We will also extract some device-specific information because our application program in Chapter 6 will need to search on this.

We'll follow the same steps as in the first example, but there is less work to do.

Example 2B—Step 1: Human Interface Design

This example is a display-only program, so we don't need buttons, etc. The value of this program is the data collected; this is displayed in a window (Figure 6-15). For each HID discovered, the program lists, if present, the VendorID and Manufacturer's name and the ProductID and Product name.

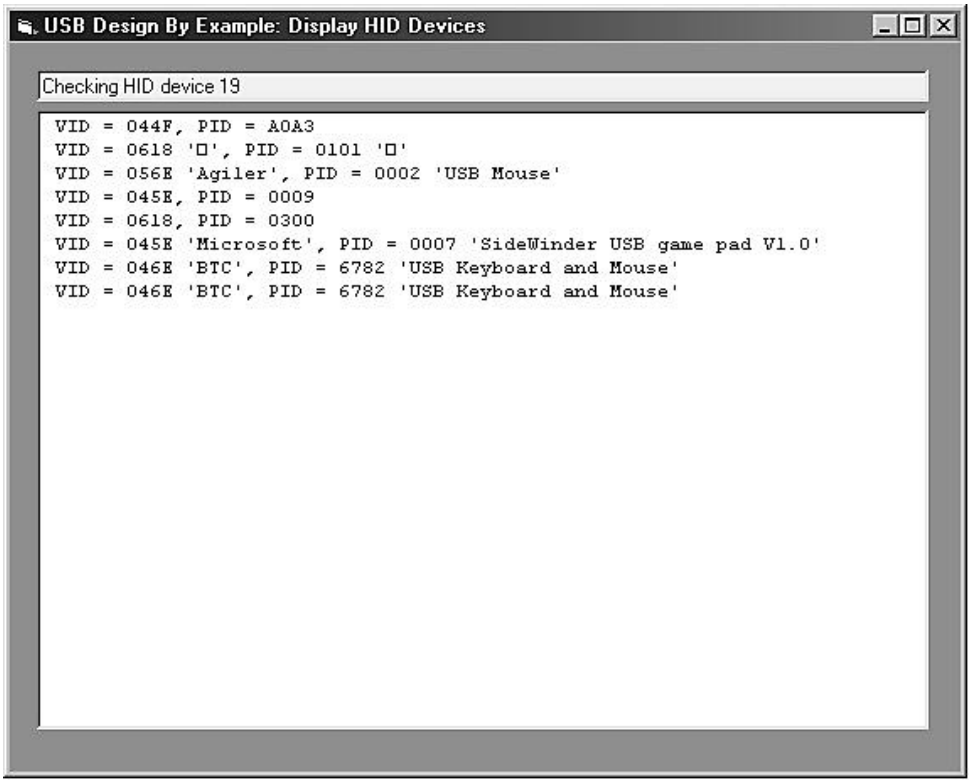

Figure 6-15. The first, and only, form

Example 2B—Step 2: Initializing Program

Because there is no user interaction in this program, all of the work can be done in the Form_Load subroutine. We start searching for information at a system root node. The currently active I/O devices are listed at the system Plug and Play node, so we open a connection to this node, selecting information on HIDs (Figure 6-16).

```
Private Sub Form_Load()
'   Look for HIDs.
'
Dim hidGuid As Guid
'
'   First, get my class identifier
Call HidD_GetHidGuid(hidGuid.Data(0))
'
'   Get a handle for the Plug and Play node, request currently active HIDs
PnPHandle& = SetupDiGetClassDevs(hidGuid.Data(0), 0, 0, &H12)
```

Figure 6-16. Connecting to the Plug and Play node

The operating system will return a list of devices with as many entries as there are active devices. We must search from the beginning of the list looking for valid entries. Once a valid entry is identified, we can request the system name relating to this entry. A two-stage process is required to get the system name: The first call returns the length of the name, and the second returns a byte array containing the name. This byte array must be converted to a Visual Basic string so that a connection can be opened to the HID. Figure 6-17 shows the code required to identify and open a HID.

```
'   Let's look for a maximum of 20 HIDs
For HIDdevice& = 0 To 19
DeviceInterfaceData.cbsize = 28 'Length of data structure in bytes
'
'   Is there an HID at this table entry
'   If SetupDiEnumDeviceInterfaces(PnPHandle&, 0, hidGuid.Data(0), HIDdevice&, _
          DeviceInterfaceData.cbsize) Then
'   There is a device here, find out its system name.  Get the length of the name first
      RequiredLength& = 0
      success = SetupDiGetDeviceInterfaceDetail(PnPHandle&, DeviceInterfaceData.cbsize, _
          0, 0, RequiredLength&, 0)
'   Now get the name itself
      PredictedLength& = RequiredLength&
      FunctionClassDeviceData.cbsize = 5
      success = SetupDiGetDeviceInterfaceDetail(PnPHandle&, DeviceInterfaceData.cbsize, _
          AddressFor(FunctionClassDeviceData.cbsize), PredictedLength&, ReturnedLength&, 0)
' Convert data array to Visual Basic String
      Temp$ = "": i& = 0: Do While FunctionClassDeviceData.DataPath(i&) <> 0
      Temp$ = Temp$ & Chr(FunctionClassDeviceData.DataPath(i&)): i& = i& + 1: Loop
' Can now open this HID
      HidHandle& = CreateFile(Temp$, &HC0000000, 3, 0, 3, 0, 0)
```

Figure 6-17. Opening a connection to a HID

Example 2B—Step 3: Displaying HID Information

Once we have a handle to a human interface device, getting information is straightforward because Microsoft has done all of the work already. The HID.DLL program contains subroutines to access the HID information, so all we have to do is make a library call and the desired information is placed in a buffer for us! Figure 6-18 lists the available function calls in HID.DLL that we can use. The first half of the list deals with Descriptor information, and the second half deals with the HID Report mechanism that is presented later in the book.

```
HidD_GetAttributes - returns VendorID, ProductID, Version#
HidD_GetManufacturerString
HidD_GetProductString
HidD_GetIndexedString
HidD_GetSerialNumberString
HidD_GetConfiguration
HidD_SetConfiguration
HidD_GetPreparsedData
HidD_FreePreparsedData
HidD_GetFeature
HidD_SetFeature
HidD_GetNumInputBuffers
HidD_SetNumInputBuffers
HidD_GetPhysicalDescriptor
```

Figure 6-18. HID.DLL is used to access HID data

Example 2B—Summary

Working with human interface devices is simple because there is a lot of support within the operating system (I used the Windows operating system in my example, but other operating systems have similar capabilities). Our HID display example was easy because of the HID support within the operating system. Windows supports the keyboard and mouse as HIDs, and the HID.DLL library provides access to the same software drivers.

Example 2C – Display HID devices

The Visual C version of Display HID devices is implemented as a Win32app. The Visual C++ wizard generates the program template and we add the functionality. The data collection takes several seconds and I found it disconcerting waiting for the main window to open so I decided to display the main window early and wait for the user before collecting and displaying the HID information. After a sign-on message the user is prompted to select NEW to start the HID information display.

The data collection uses the same algorithm as example 2B as shown in Figure 6-19

<<This segment will be added later>>

Figure 6-19 Collecting HID information

CHAPTER SUMMARY

This chapter has demonstrated the key pieces of PC host software required to access USB devices. The applications program examples were written in both Visual Basic and Visual C++ to satisfy a larger audience. If "low-level" access is required, the approach taken by Example 1 is appropriate. The "high-level" approach taken by Example 2 is simpler because most of the required software is supplied in an operating system library, and all we need to do is call the appropriate routines.